Milestone Report 1: Secure User Authentication System

Project Title: User Authentication with PostgreSQL & JWT
Developer: Chitirala giridhar
Tech Stack: PostgreSQL, FastAPI, JWT, fast api.

Git link:- https://github.com/chitiralagiridhar404/bragboard.git

---

1.1a: Create Database Models for Users

Work Completed:

- Designed a user model using **PostgreSQL**.
- Defined fields:
  - user_id (Primary Key)
  - name
  - email (Unique)
  - password (Hashed)
- Implemented:
  - Email format validation.
  - Password strength checks.
  - Unique constraint on email field.

Screenshot:

- Included schema diagram showing the structure of the users table.

Verification:

- Created sample users to test:
  - Email uniqueness.
  - Validation logic.
  - Database constraints.

1.1b: Implement User Registration/Login with JWT

Work Completed:

- Developed REST API endpoints:
    - /register – for new user sign-up.
    - /login – for authentication.
- Integrated **JWT (JSON Web Tokens)** for secure session handling.
- Added middleware to:
    - Verify JWTs.
    - Protect private routes.
- Used **bcrypt** to hash and store passwords securely.

Screenshot:

- Included UI components for login/register.
- JWT token generation and response preview.

Verification:

- Registered and logged in test users.
- Verified:
    - JWT creation and decoding.
    - Route protection using token middleware.

User Login System – Deep Dive

🧱 **1. Database Setup (PostgreSQL)**

Your users table should include:

- id (Primary Key)
- name
- email (Unique)
- hashed_password

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100) UNIQUE NOT NULL,
  hashed_password TEXT NOT NULL
);
```

---

2. Password Hashing with bcrypt

Never store plain-text passwords. Use bcrypt to hash passwords during registration and verify them during login.

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_password(password: str) -> str:
  return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str) -> bool:
  return pwd_context.verify(plain_password, hashed_password)
```

---

3. Login Endpoint Logic (FastAPI)

```
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel
from sqlalchemy.orm import Session
from your_database import get_db, UserModel
from your_auth import verify_password, create_access_token
```

```
app = FastAPI()


class LoginRequest(BaseModel):
    email: str
    password: str


@app.post("/login")
def login_user(request: LoginRequest, db: Session = Depends(get_db)):
    user = db.query(UserModel).filter(UserModel.email == request.email).first()
    if not user or not verify_password(request.password, user.hashed_password):
        raise HTTPException(status_code=401, detail="Invalid credentials")

    token = create_access_token(data={"sub": user.email})
    return {"access_token": token, "token_type": "bearer"}
```

---

4. JWT Token Generation

```
from jose import JWTError, jwt
from datetime import datetime, timedelta

SECRET_KEY = "your_secret_key"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

def create_access_token(data: dict):
    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

---

5. Protecting Routes with JWT Middleware

```
from fastapi.security import OAuth2PasswordBearer
from fastapi import Depends

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login")

def get_current_user(token: str = Depends(oauth2_scheme)):
  try:
    payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
    email = payload.get("sub")
    if email is None:
      raise HTTPException(status_code=401, detail="Invalid token")
    return email
  except JWTError:
    raise HTTPException(status_code=401, detail="Token verification failed")
```

Use Depends(get_current_user) in any route to protect it.

---

6. Verification Steps

- Try logging in with valid and invalid credentials.
- Check token expiration and decoding.
- Test protected routes with and without valid tokens.

Week 1 report
submitted by -chitirala Giridhar.